Reachability in Distributed Memory Automata

Benedikt Bollig
CNRS, LSV, ENS Paris-Saclay, Université Paris-Saclay, Gif-sur-Yvette, France
bollig@lsv.fr

Fedor Ryabinin
IMDEA Software Institue, Madrid, Spain
fedor.ryabinin@imdea.org

Arnaud Sangnier
IRIF, Université de Paris, CNRS, France
sangnier@irif.fr

Abstract

We introduce Distributed Memory Automata, a model of register automata suitable to capture some features of distributed algorithms designed for shared-memory systems. In this model, each participant owns a local register and a shared register and has the ability to change its local value, to write it in the global memory and to test atomically the number of occurrences of its value in the shared memory, up to some threshold. We show that the control-state reachability problem for Distributed Memory Automata is PSPACE-complete for a fixed number of participants and is in PSPACE when the number of participants is not fixed a priori.

2012 ACM Subject Classification Theory of computation → Concurrency

Keywords and phrases Distributed algorithms, Atomic snapshot objects, Register automata, Reachability

Digital Object Identifier 10.4230/LIPIcs.CSL.2021.13

Related Version A full version of the paper is available at `https://hal.archives-ouvertes.fr/hal-02983089`.

Funding Partly supported by ANR FREDDA (ANR-17-CE40-0013).

1 Introduction

Distributed algorithms are nowadays building blocks of modern systems in almost all computer-aided areas. One can find them in ad-hoc networks, telecommunication protocols, cache-coherence protocols, swarm robotics, or biological models. Such systems often consist of small components that solve subtasks such as mutual exclusion, leader election, or spanning trees [9, 12].

One way to classify distributed algorithms is according to how processes communicate with each other. Among the most popular classes are message-passing algorithms or shared-memory systems. In the latter case, processes write to a global memory that can be read by other processes. An important instance of a global memory are atomic snapshot objects, where every process has a dedicated global memory cell it can write to and, as the name suggests, can "snapshot" the current state of *all* global memory cells. Snapshot objects are exploited in renaming algorithms whose aim is to assign to every process a unique id from a small[1] namespace [6]. In a snapshot algorithm, every process may choose a value that is currently not in the global memory, and write it in its local memory. These two steps are non-atomic so that, in principle, other processes may simultaneously choose the same

[1] but unbounded, as it may depend on the number of processes

value. A process may then examine the snapshot (for example, check whether it contains its local value) and decide how to proceed (for example, overwrite its global memory cell by the contents of its local memory cell).

In view of their widespread use, distributed algorithms are often subject to strong correctness requirements. However, they are inherently difficult to verify. One reason is that they are usually designed for an unbounded number of participants manipulating data from an unbounded domain. That is, we have to deal with two sources of infinity during their analysis. In this paper, we take a further step towards the modeling and verification of algorithms involving atomic snapshot objects.

The Model. We introduce *distributed memory automata (DMAs)*, which feature some of the above-mentioned communication primitives of snapshot objects. Our model is based on *register automata*, which have been used as a general formal model of systems that involve (unbounded or infinite) data. Register automata go back to the work of Kaminski and Francez [10] and have recently sparked new interest leading to extensions with various applications [2, 4, 7, 13]. In a network of a DMA, every process is equipped with two registers, one representing its local memory cell, and one representing its global memory cell that every other process can read. Just like register automata, we allow registers to carry data values, i.e., values from an infinite domain (such as process identifiers), albeit comparison is only possible wrt. equality. Both, write and read operations, are restricted though. A process can perform three types of actions, which are all inspired by snapshot algorithms. It may (i) write a new value, currently not present in any global register, into its local register, (ii) copy the value from its local into its global register, and (iii) test how often its local value already occurs in the overall global memory. Note that (i) and (iii) indeed correspond to a scan operation followed by a test in atomic-snapshot algorithms. Variants of register automata have already been used to model distributed algorithms, but in a round-based setting with peer-to-peer communication [1, 5], whereas DMAs can be classified as asynchronous shared-memory systems.

Parameterized Verification. The vast majority of register-automata models impose a bound on the number of registers. In the execution of a DMA, on the other hand, the number of registers is not fixed in advance: it is *parameterized*. Indeed, distributed algorithms are often characterized by the fact that they run on systems with any number of, a priori identical, processes. Since, in many applications, the number of components varies or is unknown, these algorithms must be working on an architecture of any size. Such systems are called *parameterized*, where the parameter is the number of processes or components. Just like register automata, parameterized verification has had a long history and continues to be an active research area. We refer to [3, 8] for overviews.

In this paper, we consider a simple reachability question for DMAs, which amounts to safety verification (is a "bad" control state reachable?). In general, there are (at least) two ways to analyze parameterized systems. In the "fixed-process case", we know in advance how many processes are involved. This problem often reduces to solving reachability questions in standard models. The parameterized reachability problem, on the other hand, asks whether a given control state is reachable in some execution, involving an arbitrary number of processes. In general, this requires different techniques. Some systems, however, enjoy *cut-off* and *monotonicity* properties. In that case, the number of processes that allow for reaching a given state can be found by solving finitely many fixed-process instances [3].

Results for Distributed Memory Automata. In the fixed-process case, a standard argument allows us to restrict the problem to a bounded number of data values and to show membership in PSPACE. We also provide a matching lower bound. The PSPACE-complete intersection emptiness problem for a collection of finite-state automata is an evident starting point [11]. However, the reduction turns out to be subtle due to the fact that all processes in a DMA look the same. In particular, we have to use guards in a nested fashion to "separate" these processes so that each of them can simulate a different finite automaton.

In the case of parameterized reachability, we show that control-state reachability is in PSPACE, too, leaving tightness of this upper bound as an open problem. The proof proceeds in two steps. We first show PSPACE membership of a "subproblem", which we name *train reachability*. As a model of shared ressources with a parameterized number of processes, it is of independent interest. This algorithm is then called repeatedly within a saturation procedure that allows us to gradually compute the set of all reachable control states.

Outline. The paper is organized as follows. In Section 2, we define our model of DMAs. In Section 3, we consider the case of a fixed number of processes, for which control-state reachability is PSPACE-complete. We then move on to the case of a parameterized number of processes. The proof spans over two sections: In Section 4, we introduce and solve parameterized train reachability. This is exploited, in Section 5, to show decidability, and PSPACE membership, for parameterized reachability in DMAs. Missing proofs can be found in the long version of the paper, available at `https://hal.archives-ouvertes.fr/hal-02983089`.

2 Reachability in Distributed Memory Automata

We start with a few preliminary definitions. For $n \in \mathbb{N}$, we let $[0, n] := \{0, \dots, n\}$ and $[1, n] := \{1, \dots, n\}$. For a set A, a natural number $n \geq 1$, a tuple $\mathbf{a} \in A^n$, and $i \in [1, n]$, we let $\mathbf{a}[i]$ refer to the i-th component of $\mathbf{a}$. For $d \in A$, we let $|\mathbf{a}|_d = |\{i \in [1, n] \mid \mathbf{a}[i] = d\}|$ denote the number of occurrences of d in $\mathbf{a}$. Accordingly, we write $d \in \mathbf{a}$ if $|\mathbf{a}|_d \geq 1$, and $d \notin \mathbf{a}$ if $|\mathbf{a}|_d = 0$.

Suppose we have a system with $n \geq 1$ processes. Processes are referred to by their index $p \in [1, n]$. In the global memory, every process has a dedicated memory cell, holding a natural number (which may be a process identifier, a sequence number, etc.). Thus, the state of the global memory is a tuple $\mathbf{M} \in \mathbb{N}^n$. Similarly, every process has a local memory cell. The contents of all local memory cells is also described by a tuple $\boldsymbol{\ell} \in \mathbb{N}^n$. A process p can take a snapshot of the global memory $\mathbf{M}$ and examine its contents. More precisely, p can

- test how often its local value $\boldsymbol{\ell}[p]$ occcurs in $\mathbf{M}$, up to some threshold,
- modify its local memory cell by assigning it *some* new value that is currently not present in *the whole of* $\mathbf{M}$, or
- modify its global memory cell by assigning it its local value (and thus overwriting the old value of $\mathbf{M}[p]$).

Accordingly, $\mathcal{T} = \{=_t, <_t, >_t \mid t \in \mathbb{N}\}$ is the set of *tests* and $\Sigma = \{\mathsf{new}, \mathsf{write}\} \cup \mathcal{T}$ the set of *actions*. For $k \in \mathbb{N}$ and $\bowtie_t \in \mathcal{T}$ with $\bowtie \in \{=, <, >\}$, we write $k \models \bowtie_t$ if $k \bowtie t$. We are now prepared to define distributed memory automata.

▶ **Definition 1.** *A* distributed memory automaton (DMA) *is a tuple* $\mathcal{A} = (S, \iota, \Delta, F)$ *where* S *is the finite set of* states, $\iota \in S$ *is the* initial state, $\Delta \subseteq S \times \Sigma \times S$ *is the finite set of* transitions, *and* F *is the set of* final states.

For a test $\bowtie_t \in \mathcal{T}$, we let $|\bowtie_t| = \max\{1, t\}$. Moreover, $|\mathsf{new}| = |\mathsf{write}| = 1$. The size of $\mathcal{A}$ is defined as $|\mathcal{A}| := |S| + \sum_{(s,\sigma,s')\in\Delta} |\sigma|$. Note that we assume a unary encoding of tests.

For $n \geq 1$, an n-configuration (shortly a configuration) is a tuple $\gamma = (\mathbf{s}, \boldsymbol{\ell}, \mathbf{M}) \in S^n \times \mathbb{N}^n \times \mathbb{N}^n$. Given a process $p \in [1, n]$, we consider that $\mathbf{s}[p]$ is the current state of p, $\boldsymbol{\ell}[p]$ is the content of its local memory, and $\mathbf{M}[p]$ is the entry of p in the global memory. We use $states(\gamma)$ to denote the set $\{\mathbf{s}[p] \mid p \in [1, n]\}$ and $|\gamma|$ to represent the number of processes n of the configuration γ.

We say that γ is *initial* if, for all $p \in [1, |\gamma|]$, we have $\mathbf{s}[p] = \iota$ and $\boldsymbol{\ell}[p] \notin \mathbf{M}$, and for all $p, q \in [1, |\gamma|]$, $\boldsymbol{\ell}[p] = \boldsymbol{\ell}[q]$ implies $p = q$. Hence, in an initial configuration, each process has a different value in its local register and none of these values appears in the shared memory. Moreover, configuration γ is called *final* if $\mathbf{s}[p] \in F$ for some $p \in [1, |\gamma|]$, i.e., if one of its processes is in a state of F.

Let $\mathbb{C}_{\mathcal{A},n}$ be the set of n-configurations and $\mathbb{C}_{\mathcal{A}} := \bigcup_{n\geq 1} \mathbb{C}_{\mathcal{A},n}$ be the set of all configurations. We define a global transition relation $\Longrightarrow_{\mathcal{A}} \subseteq \mathbb{C}_{\mathcal{A}} \times (\Sigma \times \mathbb{N}) \times \mathbb{C}_{\mathcal{A}}$. Suppose $\gamma = (\mathbf{s}, \boldsymbol{\ell}, \mathbf{M})$ and $\gamma' = (\mathbf{s}', \boldsymbol{\ell}', \mathbf{M}')$ are two configurations and let $\sigma \in \Sigma$ and $p \in [1, |\gamma|]$. We let $\gamma \xRightarrow{(\sigma,p)}_{\mathcal{A}} \gamma'$ if the following hold:

- $|\gamma| = |\gamma'|$ and
- $(\mathbf{s}[p], \sigma, \mathbf{s}'[p]) \in \Delta$,
- $\mathbf{s}[q] = \mathbf{s}'[q]$ and $\boldsymbol{\ell}[q] = \boldsymbol{\ell}'[q]$ and $\mathbf{M}[q] = \mathbf{M}'[q]$ for all $q \in [1, |\gamma|] \setminus \{p\}$,
- if $\sigma = \mathsf{new}$, then $\boldsymbol{\ell}'[p] \notin \mathbf{M}$ and $\mathbf{M} = \mathbf{M}'$,
- if $\sigma = \mathsf{write}$, then $\boldsymbol{\ell}[p] = \boldsymbol{\ell}'[p] = \mathbf{M}'[p]$,
- if $\sigma \in \mathcal{T}$, then $\boldsymbol{\ell} = \boldsymbol{\ell}'$ and $\mathbf{M} = \mathbf{M}'$ and $|\mathbf{M}|_{\boldsymbol{\ell}[p]} \models \sigma$.

We write $\Longrightarrow_{\mathcal{A}}$ for the union of all relations $\xRightarrow{(\sigma,p)}_{\mathcal{A}}$ and denote by $\Longrightarrow^*_{\mathcal{A}}$ the reflexive and transitive closure of $\Longrightarrow_{\mathcal{A}}$. Note that if $\gamma \Longrightarrow_{\mathcal{A}} \gamma'$ then there exists $n \geq 1$ such that $\gamma, \gamma' \in \mathbb{C}_{\mathcal{A},n}$. In fact, the transition relation $\Longrightarrow_{\mathcal{A}}$ does not change the number of involved processes. If we have $(\mathbf{s}, \boldsymbol{\ell}, \mathbf{M}) \xRightarrow{(\mathsf{new},p)}_{\mathcal{A}} (\mathbf{s}', \boldsymbol{\ell}', \mathbf{M}')$ with $\boldsymbol{\ell}'[p] = d$, we will sometimes write $(\mathbf{s}, \boldsymbol{\ell}, \mathbf{M}) \xRightarrow{(\mathsf{new}(d),p)}_{\mathcal{A}} (\mathbf{s}', \boldsymbol{\ell}', \mathbf{M}')$ to provide explicitly the new local value. A *run* ρ of $\mathcal{A}$ is a finite sequence of the form $\gamma_0 \xRightarrow{(\sigma_0,p_0)}_{\mathcal{A}} \gamma_1 \xRightarrow{(\sigma_1,p_1)}_{\mathcal{A}} \gamma_2 \cdots \xRightarrow{(\sigma_{k-1},p_{k-1})}_{\mathcal{A}} \gamma_k$ where $\gamma_i \in \mathbb{C}_{\mathcal{A}}$ for all $i \in [0, k]$ and γ_0 is initial. It is said to be final if γ_k is final.

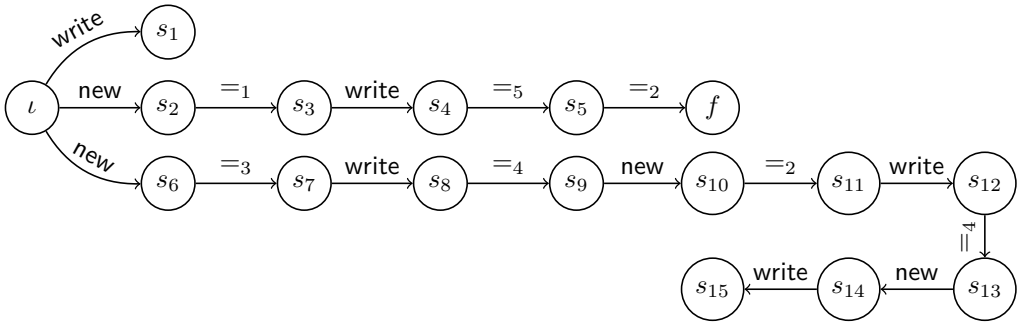


Figure 1 An example DMA.

▶ **Example 2.** In the example presented in Figure 1, the final state f is reachable and we shall see in the development of the paper how we can prove this, since it is not obvious at first sight. We present here an execution to reach s_9 with four processes. Assume that the initial configuration is $\big([\iota, \iota, \iota, \iota], [0, 1, 2, 3], [4, 4, 4, 4]\big)$. From this configur-

ation, if one process performs a write going to s_1, then the system will not be able to reach s_9, because no other processes will be able to choose the same value (with a new) since the value is written in the global memory and the consecutive test $=_4$ (necessary to reach s_9) will never be available. Instead, to reach s_9, we perform the following step: $\big([\iota,\iota,\iota,\iota],[0,1,2,3],[4,4,4,4]\big) \xrightarrow{(\mathsf{new},2)}_{\mathcal{A}} \big([\iota,s_2,\iota,\iota],[0,0,2,3],[4,4,4,4]\big)$. Here the second process can choose the same local value as the first one since it is not yet written in the memory. Thanks to the sequence $\xrightarrow{(\mathsf{new},3)}_{\mathcal{A}}\xrightarrow{(\mathsf{new},4)}_{\mathcal{A}}\xrightarrow{(\mathsf{write},1)}_{\mathcal{A}}$, we reach the configuration $\big([s_1,s_2,s_2,s_6],[0,0,0,0],[0,4,4,4]\big)$, from which we can perform the transition sequence $\xrightarrow{(=_1,2)}_{\mathcal{A}}\xrightarrow{(=_1,3)}_{\mathcal{A}}\xrightarrow{(\mathsf{write},2)}_{\mathcal{A}}\xrightarrow{(\mathsf{write},3)}_{\mathcal{A}}$ to reach the configuration $\big([s_1,s_4,s_4,s_6],[0,0,0,0],[0,0,0,4]\big)$ from which it is possible to perform $\xrightarrow{(=_3,4)}_{\mathcal{A}}\xrightarrow{(\mathsf{write},4)}_{\mathcal{A}}\xrightarrow{(=_4,4)}_{\mathcal{A}}$ making the fourth process reach s_9. Note that we could build a similar execution with 5 processes to reach the configuration $\big([s_1,s_4,s_4,s_4,s_9],[0,0,0,0,0],[0,0,0,0,0]\big)$ by adding an extra process that behaves as process two but writes its value after the last process reaches s_9. We have then five times the value 0 in the global memory. But from this configuration, it is not possible to reach f since, to pass the sequence of transitions $(s_4,=_5,s_5),(s_5,=_2,f)$, at least three processes have to delete the value 0 from their global memory and this is not possible.

The main problem we study in the paper is the reachability problem, in which we check whether a state of a given DMA can be reached without specifying the number of processes. In other words, the number of processes is a parameter that needs to be instantiated.

Reachability	
I:	DMA $\mathcal{A}$
Q:	$\gamma \Longrightarrow_{\mathcal{A}}^{*} \gamma'$ for some initial $\gamma \in \mathbb{C}_{\mathcal{A}}$ and some final $\gamma' \in \mathbb{C}_{\mathcal{A}}$?

In order to understand the above problem, it is important to also know how to solve the respective problem where the number of processes is imposed.

Fixed-Reachability	
I:	DMA $\mathcal{A}$ and $n \geq 1$ (encoded in unary)
Q:	$\gamma \Longrightarrow_{\mathcal{A}}^{*} \gamma'$ for some initial $\gamma \in \mathbb{C}_{\mathcal{A},n}$ and some final $\gamma' \in \mathbb{C}_{\mathcal{A},n}$?

Hence, Reachability consists in checking the existence of a final run and Fixed-Reachability seeks for a final run with an initial n-configuration.

3 Considering a fixed number of processes

In this section, we show that Fixed-Reachability is Pspace-complete.

First we explain how we obtain the upper bound. We consider a DMA $\mathcal{A} = (S, \iota, \Delta, F)$ and a fixed number of processes $n \geq 1$. Note that, for any configuration $\gamma = (\mathbf{s}, \boldsymbol{\ell}, \mathbf{M}) \in S^n \times \mathbb{N}^n \times \mathbb{N}^n$, the number of different values in the local memory $\boldsymbol{\ell}$ and in the global memory $\mathbf{M}$ is at most $2n$. Hence, if there is a run $\gamma_0 \xrightarrow{(\sigma_0,p_0)}_{\mathcal{A}} \gamma_1 \xrightarrow{(\sigma_1,p_1)}_{\mathcal{A}} \gamma_2 \cdots \xrightarrow{(\sigma_{k-1},p_{k-1})}_{\mathcal{A}} \gamma_k$ such that $\gamma_i \in \mathbb{C}_{\mathcal{A},n}$ for all $i \in [0,k]$ and γ_0 is initial and γ_k is final, then there is a run $\gamma'_0 \xrightarrow{(\sigma_0,p_0)}_{\mathcal{A}} \gamma'_1 \xrightarrow{(\sigma_1,p_1)}_{\mathcal{A}} \gamma'_2 \cdots \xrightarrow{(\sigma_{k-1},p_{k-1})}_{\mathcal{A}} \gamma'_k$ such that $\gamma'_i \in S^n \times [0,2n]^n \times [0,2n]^n$ for all $i \in [0,k]$ and γ'_0 is initial and γ'_k is final. In fact, the set of values $[0,2n]^n$ is enough to define an initial configuration in $\mathbb{C}_{\mathcal{A},n}$ since we can pick $2n$ different values. Since there are $2n+1$ different values in $[0,2n]$, when performing an action new, it is always possible to pick

a value in $[0, 2n]$ that appears neither in the local memory nor in the global memory. To solve Fixed-Reachability for n processes, we then check whether a final configuration is reachable from an initial one in the graph where the set of vertices is $S^n \times [0, 2n]^n \times [0, 2n]^n$ and the edges are defined by the transition relation $\Longrightarrow_{\mathcal{A}}$. This graph having an exponential number of vertices, the search can be performed in NPspace, i.e., in Pspace thanks to Savitch's theorem. Note that we could obtain the same upper bound by reducing our problem to the non-emptiness problem for non-deterministic register automata with $2n$ registers and S^n as a set of states and then use the fact that the non-emptiness problem for such automata is in Pspace [7]. The $2n$ registers will correspond to the local and global memory and the different actions of the DMA can be simulated by a register automaton.

▶ **Proposition 3.** *Fixed-Reachability is in Pspace.*

To show the lower bound, we do a reduction from the intersection emptiness problem of many non-deterministic finite state automata. A non-deterministic finite state automaton (FSA) A over a finite alphabet Λ is a tuple (Q, q_ι, δ, F) where Q is a finite set of states, $q_\iota \in Q$ is an initial state, $\delta \subseteq Q \times \Lambda \times Q$ is the transition relation and $F \subseteq Q$ is the set of accepting states. A finite word $w = w_0 w_1 \dots w_{k-1}$ in Λ^* is accepted by A if there exists a sequence of states $(q_i)_{0 \leq i \leq k}$ such that $q_0 = q_\iota$, $q_k \in F$, and $(q_i, w_i, q_{i+1}) \in \delta$ for all $i \in [0, k-1]$. We denote by $\mathcal{L}(A)$ the language of A, i.e., the set of words $\{w \in \Lambda^* \mid w \text{ is accepted by } A\}$. The emptiness intersection problem asks, given m FSA $A_1, \dots, A_m$ over the alphabet Λ, whether $\bigcap_{1 \leq i \leq m} \mathcal{L}(A_i) = \emptyset$. This problem is known to be Pspace-complete [11].

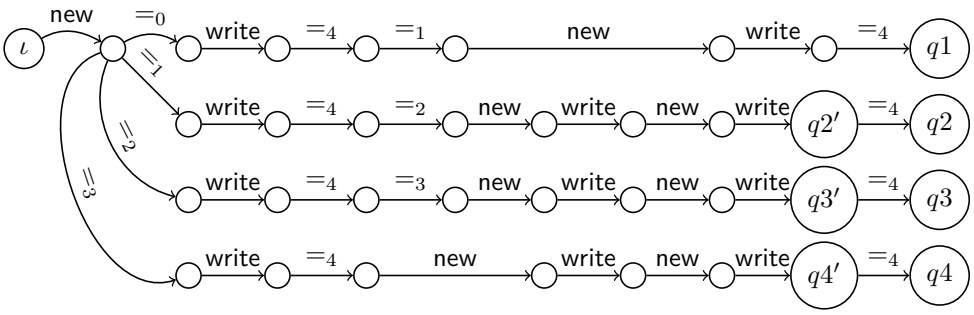
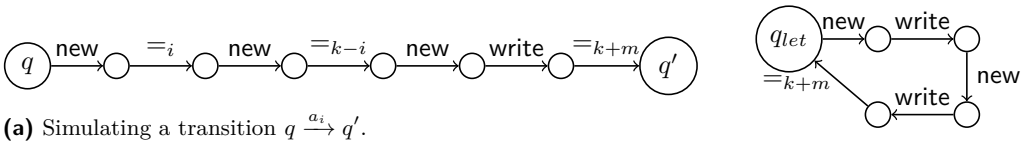


Figure 2 Gadget to isolate 4 processes.

In order to reduce the intersection emptiness problem for FSA to Fixed-Reachability, we first need a gadget to bring different processes to different parts of the DMA so that each of these processes can simulate a particular finite automaton. This gadget is necessary since in DMA all processes begin in the same initial state. An example of this gadget for four processes is depicted in Figure 2. At the beginning, all the processes are in the initial state ι and we claim that if a process reaches the state $q1$ then there is one process in $q2$ or in $q2'$ (because at this stage we cannot force the transition labeled with the test $=_4$ leading to $q2$ to be taken), one process in $q3$ or in $q3'$ and one process in $q4$ or in $q4'$. In fact, if one process is in $q1$, then it has to first write its local value and the only way to do this is to take the upper branch of the DMA and, after writing, wait for the other processes to write their value in order to pass the test $=_4$. Because of this test, all the processes have to choose the same value with the first new. One way to pass the test $=_4$ for the first process is that all the processes take the upper branch as follows: they all choose the same new value, then they all pass the test $=_0$ then they all write their value and they all pass the test $=_4$. However this execution will then stop because of the following test $=_1$ which could not be taken because,

at this stage, none of the processes can rewrite its value in the global memory. The same reasoning can be iterated to show that the only way to pass the test $=_1$ in the upper branch is to have one process per branch, the first one writes its value, then the second one can pass the first test $=_1$ in the second branch and writes the same value, the third one passes the test $=_2$ in the third branch and writes its value and the last one can pass the test $=_3$ in the last branch and writes its value. Each process can then pass, in its branch, the test $=_4$ but only the fourth process can perform a new followed by a write to overwrite its value in the global memory (the other ones have to wait because of the tests $=_3, =_2, =_1$). Hence the fourth process overwrites its value, then the third one, then the second one and finally the first process can pass the test $=_1$. After that all the processes can again perform a new and write to choose the same new value and write it to the memory to allow the first process to reach $q1$.

We consider now an instance of the intersection emptiness problem with m FSA $A_i = (Q_i, q_\iota^{(i)}, \delta_i, F_i)$ for $i \in [1, m]$ working over the finite alphabet $\Lambda = \{a_1, a_2, \ldots, a_k\}$. Without loss of generality, we can assume that, for each $i \in [1, m]$, the set $F_i = \{q_f^{(i)}\}$ is a singleton and furthermore the only way to reach this state is to read the letter a_k that is not present in any other transitions. Hence all the words accepted by A_i end with a_k and if an automaton reads a word until its last letter a_k, then the automaton accepts this word.



(a) Simulating a transition $q \xrightarrow{a_i} q'$.

(b) Simulating the k letters.

Figure 3 Encoding intersection emptiness of finite automata into DMA.

To check whether $\bigcap_{1 \le i \le m} \mathcal{L}(A_i) = \emptyset$, we build a DMA and consider $m + k$ processes. The first m processes simulate the automata $(A_i)_{1 \le i \le m}$ and the k last processes simulate the read letters. First we use the gadget presented previously to separate these $m + k$ processes in different parts of the DMA. For $i \in [1, m]$, the i-th process will be brought to the initial state $q_\iota^{(i)}$ of each NFA whereas the last k processes are brought to the state q_{let} leading to the part of the DMA depicted in Figure 3b.

We show then on Figure 3a how we simulate each transition $q \xrightarrow{a_i} q'$ of the finite state automata in the DMA. A process $p \in [1, m]$, in order to simulate the transition $q \xrightarrow{a_i} q'$, first takes a new value and waits until this value appears i times in the global memory. At this stage only the k last processes are able to write, so i of these last processes take the same new value and write it to the global memory. There possibly remain at most $k - i$ processes that did not take the same new value. But the process p then takes a new value and it has to appear $k - i$ times in the global memory, so the $k - i$ processes that did not write their value to the memory can do it now. Finally, after this, each process can take a new value and write it to the global memory and if they all have taken the same new value, they can all pass the test $=_{k+m}$. This ensures that all the processes simulating the automata have read the same letter and, moreover, that the different processes are synchronized. For instance, imagine that a process simulating the automaton takes the transitions $\xrightarrow{=_1}\xrightarrow{\text{new}}\xrightarrow{=_{k-1}}$ and another one at the same stage of the simulation goes through $\xrightarrow{=_2}\xrightarrow{\text{new}}\xrightarrow{=_{k-2}}$. This is possible: a process $p1$ simulating a letter writes its value to the memory allowing the test $=_1$, then a second process simulating a letter writes the same value to the memory allowing the test $=_2$, then the $k - 2$ remaining last processes take the same new value and so does the process $p1$

(by taking the third transition labelled by new in the loop starting in q_{let}), then the $k-2$ last processes write their value allowing the test $=_{k-2}$ and finally the process $p1$ writes its value allowing the test $=_{k-1}$. But after this, the different processes are blocked because $p1$ cannot take a new value anymore and write it to allow the test $=_{k+m}$ for which all the processes need to choose the same new value and write it to the global memory.

To finalize our reduction we choose $\{q_f^{(1)}\}$ as the set of final states of the DMA. Since the size of the DMA we build is polynomial in the size of the m automata, we can deduce the lower bound for Fixed-Reachability.

▶ **Theorem 4.** *Fixed-Reachability is Pspace-complete.*

4 The parameterized train problem

We introduce in this section a simpler parameterized problem whose resolution will help in solving the reachability problem in DMA.

4.1 Definition

As for DMA, we will use here the set of tests $\mathcal{T} := \{=_t, <_t, >_t \mid t \in \mathbb{N}\}$. Our problem consists in modelling a set of passengers who can enter a train and leave it. Each passenger enters the train at most once and has the ability to test how many passengers are in the train and to change its state accordingly. Furthermore, there is a distinguished passenger, called the *controller*.

▶ **Definition 5.** *A* train automaton *is a tuple* $TA = (S, \iota^c, \iota, S_{out}, S_{in}, \Delta, s_f)$ *where* S *is the finite set of* states *partitioned into* $S = S_{out} \uplus S_{in} \uplus \{s_f\}$, $\iota^c \in S_{out}$ *is the* initial state *for the controller,* $\iota \in S_{out}$ *is the* initial state *for the passengers,* s_f *is the final state, and* $\Delta \subseteq (S_{out} \times \mathcal{T} \times S_{out}) \cup (S_{in} \times \mathcal{T} \times S_{in}) \cup (S_{out} \times \{\mathbf{E}\} \times S_{in}) \cup (S_{in} \times \{\mathbf{Q}\} \times \{s_f\})$ *is the finite set of* transitions.

Intuitively, when a passenger (or the controller) is in a state from S_{out} or in s_f, he stands outside the train, and when he is in S_{in}, he is inside the train. A passenger enters the train thanks to the action $\mathbf{E}$. He can leave the train with action $\mathbf{Q}$ and, in doing so, enters the state s_f from which he cannot perform any test or action. We now detail the semantics induced by TA.

For $n \geq 1$, an *n-train configuration* is a pair $\theta = (\mathbf{s}, c) \in S^n \times \mathbb{N}$ such that $\mathbf{s}[1]$ is the controller state and $c = |\{p \in [1, n] \mid \mathbf{s}[p] \in S_{in}\}|$. Note that we identify the controller with the first passenger. Formally, we could get rid of the c since we can deduce it from $\mathbf{s}$, but it eases the writing of our results to keep it. A *train configuration* is an n-train configuration for some $n \geq 1$. For an n-train configuration θ, we denote by $|\theta| = n$ its size. We say that θ is *initial* if $\mathbf{s}[1] = \iota^c$, $\mathbf{s}[p] = \iota$ for all $p \in [2, |\theta|]$, and $c = 0$. We define a transition relation $\to_{TA}$ as follows. Let $\theta = (\mathbf{s}, c)$ and $\theta' = (\mathbf{s}', c')$ be two train configurations, $a \in \mathcal{T} \cup \{\mathbf{E}, \mathbf{Q}\}$, and $p \in [1, |\theta|]$. We let $\theta \xrightarrow{(a,p)}_{TA} \theta'$ if $|\theta| = |\theta'|$, $\mathbf{s}[p'] = \mathbf{s}'[p']$ for all $p' \in [1, |\theta|] \setminus \{p\}$, $(\mathbf{s}[p], a, \mathbf{s}'[p]) \in \Delta$, and the following hold:

- if $a = \mathbf{E}$ then $c' = c + 1$ (passenger p enters the train),
- if $a = \mathbf{Q}$ then $c' = c - 1$ (passenger p leaves the train), and
- if $a \in \mathcal{T}$ then $c = c'$ and $c \models a$.

We write $\theta \to_{TA} \theta'$ if there exist $a \in \mathcal{T} \cup \{\mathbf{E}, \mathbf{Q}\}$ and $p \in [1, |\theta|]$ such that $\theta \xrightarrow{(a,p)}_{TA} \theta'$. An execution of TA is a finite sequence $\rho = \theta_0 \xrightarrow{(a_0,p_0)}_{TA} \theta_1 \xrightarrow{(a_1,p_1)}_{TA} \theta_2 \dots \xrightarrow{(a_{k-1},p_{k-1})}_{TA} \theta_k$ (or $\rho = \theta_0 \to_{TA} \theta_1 \to_{TA} \theta_2 \dots \to_{TA} \theta_k$ if we do not need the action and test labellings).

We denote by $\rightarrow_{TA}^*$ the reflexive and transitive closure of $\rightarrow_{TA}$. If $\theta \rightarrow_{TA}^* \theta'$, then we say that there exists an execution from θ to θ' in TA. Note that the number of passengers does not change during an execution, just like the number of processes does not change in an execution of a DMA.

The problem we study in this section can be formalized as follows:

Train-Reachability	
I:	A train automaton $TA = (S, \iota^c, \iota, S_{out}, S_{in}, \Delta, s_f)$ and a state $s \in S$
Q:	Are there an initial train configuration θ and a configuration $\theta' = (\mathbf{s}', c')$ such that $\theta \rightarrow_{TA}^* \theta'$ and $\mathbf{s}'[p] = s$ for some $p \in [1, \lvert\theta\rvert]$?

We let $\mathsf{TrainReach}(TA)$ denote the set of states $s \in S$ such the answer to the Train-Reachability with TA and s is positive.

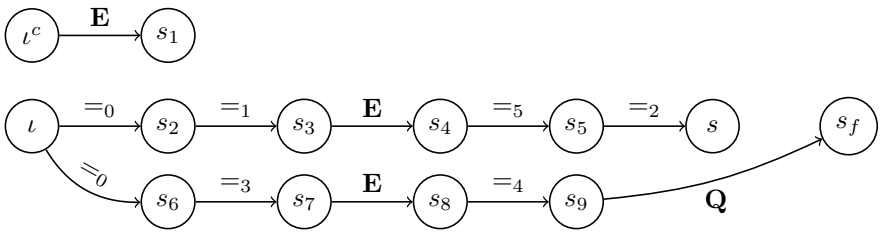


Figure 4 An example of train automaton.

▶ **Example 6.** In Figure 4, we have drawn a train automaton inspired (we shall see the connection later) from the DMA given in Figure 1. In this train automaton, the state s is not reachable. In fact, to reach it, the controller would have to go to state s_1 and at least two passengers to s_4. But then, there are at least three passengers in the train that cannot leave it anymore. Hence, the test $=_2$ can never be satisfied.

Train automata will help us to simulate part of the executions of DMAs where all the processes except one (the controller) begin by choosing a new value identical to the one of the controller (the idea being that this value corresponds to the identity of the train). Then, when a process performs a write, this corresponds to a passenger entering the train. Moreover, when, thanks to a sequence of actions, it overwrites its value in the global memory, this corresponds to a passenger leaving the train. This also explains why we need a controller in train automata: it helps to simulate a process which did not perform a new. Since, initially, all the processes have a different value in their global memory, there can be, for each value d, at most one process which did not perform a $\mathsf{new}(d)$ and has d in its local register.

4.2 Bounding the number of passengers

We will see here that in order to solve Train-Reachability, we can bound the number of passengers present in the train at any moment. Consider a train automaton $TA = (S, \iota^c, \iota, S_{out}, S_{in}, \Delta, s_f)$. We let $cap \in \mathbb{N}$ be the maximal constant appearing in the transitions of Δ. Hence we have $t \leq cap$ for all $(s, \bowtie_t, s') \in \Delta$. Given an n-train configuration $\theta = (\mathbf{s}, c)$ and a bound $b \in \mathbb{N}$, we say that θ is *b-bounded* if $c \leq b$. An execution $\theta_0 \rightarrow_{TA} \theta_1 \rightarrow_{TA} \theta_2 \ldots \rightarrow_{TA} \theta_k$ is called *b-bounded* if θ_i is b-bounded for all $i \in [0, k]$.

Finally, we introduce a relation $\preceq$ beween two train configurations $\theta = (\mathbf{s}, c)$ and $\theta' = (\mathbf{s}', c')$ defined as follows: $\theta \preceq \theta'$ if $\lvert\theta\rvert = \lvert\theta'\rvert$ and $c = c'$ and for all $p \in [1, \lvert\theta\rvert]$, if $\mathbf{s}[p] \neq \mathbf{s}'[p]$ then $\mathbf{s}[p] = s_f$ and $\mathbf{s}'[p] \in S_{out}$. In other words, if a passenger is not in the same state in θ

and in θ', it means he is in its final state in θ and he is out of the train in θ'. We need a first technical result stating that the relation $\preceq$ is a simulation relation for $\rightarrow_{TA}$. The result of this lemma is a direct consequence of the definition of $\preceq$ and of the fact that, in TA, when the controller or a passenger is in its final state, he cannot do anything anymore.

▶ **Lemma 7.** *If $\theta_1 \preceq \theta'_1$ and $\theta_1 \xrightarrow{(a,p)}_{TA} \theta_2$ then there exists a configuration θ'_2 such that $\theta_2 \preceq \theta'_2$ and $\theta'_1 \xrightarrow{(a,p)}_{TA} \theta'_2$.*

The following lemma shows us how to bound locally the capacity of the train. The idea is that if the capacity of the train goes above $cap + 2$, it is not necessary to make more passengers enter the train to satisfy the subsequent tests before the capacity goes back to a value smaller than $cap + 2$.

▶ **Lemma 8.** *Let $M > cap$. If there is an execution $\theta_0 \rightarrow_{TA} \theta_1 \rightarrow_{TA} \dots \rightarrow_{TA} \theta_k$ with $\theta_i = (\mathbf{s}_i, c_i)$ for all $i \in [0, k]$ and such that $c_0 = c_K = M$ and $c_i = M + 1$ for all $i \in [1, k-1]$, then there is an M-bounded execution from θ_0 to some θ' with $\theta_k \preceq \theta'$.*

Proof. Let $\rho = \theta_0 \xrightarrow{(a_0,p_0)}_{TA} \theta_1 \xrightarrow{(a_1,p_1)}_{TA} \theta_2 \dots \xrightarrow{(a_{k-1},p_{k-1})}_{TA} \theta_k$ be an execution with $\theta_i = (\mathbf{s}_i, c_i)$ for all $i \in [0, k]$ and such that $c_0 = c_k = M$ and $c_i = M + 1$ for all $i \in [1, k-1]$. By definition of the transition relation $\rightarrow_{TA}$ and of cap, we have necessarily $a_0 = \mathbf{E}$ and $a_{k-1} = \mathbf{Q}$ and $a_i = >_t$ with $M > cap \geq t$ for all $i \in [1, k-2]$. We distinguish two cases:

1. **Case** $p_0 = p_{k-1}$, i.e., it is the same process that enters and leaves the train. In that case, we let that process never enter the train and we consider the execution $\theta_0 \rightarrow_{TA} \theta'_1 \dots \rightarrow_{TA} \theta'_\ell = (\mathbf{s}'_\ell, c'_\ell)$, obtained from ρ by deleting all the transitions (a, p) with $p = p_0$. During this execution the number of passengers in the train remains the same and is equal to $c_0 = M$ and, for all $p \in [1, |\theta_0|] \setminus \{p_0\}$, we have $\mathbf{s}'_\ell[p] = \mathbf{s}_k[p]$ and $\mathbf{s}'_\ell[p_0] = \mathbf{s}_0[p_0]$. Since $\mathbf{s}_0[p_0] \in S_{out}$ (because at the first step of ρ the passenger p_0 enters the train) and $\mathbf{s}_k[p_0] = s_f$ (because in the last step of ρ, passenger p_0 leaves the train), we deduce that $\theta_k \preceq \theta'_\ell$.
2. **Case** $p_0 \neq p_{k-1}$. In that case, we reorder the execution ρ as follows. First we execute all the transitions (a, p) with $p = p_{k-1}$ leading to a configuration $\theta'' = (\mathbf{s}'', c'')$ such that $\mathbf{s}''[p] = \mathbf{s}_0[p]$ for all $p \in [1, |\theta_0|] \setminus \{p_{k-1}\}$ and $\mathbf{s}''[p_{k-1}] = \mathbf{s}_k[p_{k-1}] = s_f$ and $c'' = M - 1$. Then from θ'' we execute, in the same order, the remaining transition of ρ (the first being labelled with $(\mathbf{E}, p_0)$) which leads exactly to the configuration θ_k. Hence we obtain an M-bounded execution from θ_0 to θ_k. ◀

Using iteratively this last lemma allows us to bound the number of passengers in the train to reach a specific control state s.

▶ **Proposition 9.** *Let $s \in S$. Let θ be an initial configuration and $p \in [1, |\theta|]$. If there is an execution from θ to some configuration $\theta' = (\mathbf{s}', c')$ with $\mathbf{s}'[p] = s$, then there is a $(cap + 2)$-bounded execution from θ to some configuration $\theta'' = (\mathbf{s}'', c'')$ with $\mathbf{s}''[p] = s$.*

4.3 Solving Train-Reachability

We shall see now how Proposition 9 allows us to build a finite abstract graph in which the reachability problem provides us with a solution for Train-Reachability. We consider a train automaton $TA = (S, \iota^c, \iota, S_{out}, S_{in}, \Delta, s_f)$ and, as in the previous section, we let $cap \in \mathbb{N}$ be the maximal constant appearing in the transitions of Δ. In order to solve our reachability problem, we build a graph of abstract configurations which keep track of the states of the controller, of the states in S_{out} that can be reached, and of the number of people

in the train up to $cap + 2$. As we shall see, such an abstract graph will suffice to obtain a witness for Train-Reachability thanks to the Proposition 9 and to the following Copycat Lemma.

▶ **Lemma 10** (Copycat Lemma). *Let $s \in S_{out}$ and $M > 0$. Assume an M-bounded execution from an initial train configuration θ_0 to a configuration $\theta = (\mathbf{s}, c)$ with $\mathbf{s}[p] = s$ for some $p \in [2, |\theta_0|]$. Then, for all $b \geq 0$, there exists an M-bounded execution from θ'_0 to $\theta' = (\mathbf{s}', c)$ where θ'_0 is the initial train configuration with $|\theta'_0| = |\theta_0| + b$, $\mathbf{s}'[p] = \mathbf{s}[p]$ for all $p \in [1, |\theta_0|]$, and $\mathbf{s}'[p] = s$ for all $p \in [|\theta_0| + 1, |\theta_0| + b]$.*

Proof. Let $\rho = \theta_0 \xrightarrow{(a_0,p_0)}_{TA} \theta_1 \xrightarrow{(a_1,p_1)}_{TA} \theta_2 \dots \xrightarrow{(a_{k-1},p_{k-1})}_{TA} \theta_k$ be an execution with $\theta_i = (\mathbf{s}_i, c_i)$ for all $i \in [0, k]$ and $\mathbf{s}_k[p] \in S_{out}$ for $p \in [2, |\theta_0|]$. Since, in TA, a passenger can never go to a state in S_{out} once he has entered the train, $p_i = p$ implies $a_i \in \mathcal{T}$ for all $i \in [0, k-1]$. In other words, all the actions performed by passenger p along ρ are tests. Hence from θ'_0, we can reproduce ρ and each time we have $p_i = p$, passengers $|\theta_0| + 1$ to $|\theta_0| + b$ take the same transition as passenger p. As a consequence, at the end of this run, all these passengers will be in the same state as passenger p, and extending ρ in such a way is possible because the actions of passenger p never change the capacity of the train, as they are just tests. ◀

An abstract train configuration ξ of TA is a triple (s^c, Out, In) where $s^c \in S$, $Out \subseteq S_{out} \cup \{s_f\}$ and $In \in \mathbb{N}^{S_{in}}$ is a multiset of elements of S_{in} such that $\sum_{s \in S_{in}} In(s) \leq cap + 1$ if $s^c \in S_{in}$ and $\sum_{s \in S_{in}} In(s) \leq cap + 2$ otherwise. Given an abstract configuration $\xi = (s^c, Out, In)$, we define $inside(\xi) \in [0, cap + 2]$ describing the number of passengers in the train: it is equal to $\sum_{s \in S_{in}} In(s)$ if $s^c \notin S_{in}$ and $1 + \sum_{s \in S_{in}} In(s)$ otherwise. Indeed, by definition, we have $inside(\xi) \leq cap + 2$ for all abstract train configurations ξ. The initial abstract train configuration ξ_ι is then equal to $(\iota^c, \{\iota\}, In_\iota)$ with $In_\iota(s) = 0$ for all $s \in S_{in}$. We denote by Ξ the set of abstract train configurations of TA. Note that by definition Ξ is finite.

We define now a transition relation $\rightsquigarrow$ between abstract configurations. Let $\xi_1 = (s^c_1, Out_1, In_1)$ and $\xi_2 = (s^c_2, Out_2, In_2)$ be two abstract train configurations and $\delta = (s, a, s') \in \Delta$ and $mc = \{\top, \bot\}$. The value mc indicates whether the controller moves ($\top$) or another passenger ($\bot$). We have $\xi_1 \overset{\delta, mc}{\rightsquigarrow} \xi_2$ if one of the following cases holds:

1. $mc = \top$ and $s = s^c_1$ and $s' = s^c_2$ and $Out_1 = Out_2$ and $In_1 = In_2$ and if $a = \mathbf{E}$ then $inside(\xi_1) < cap + 2$ and if $a \in \mathcal{T}$ then $inside(\xi_1) \models a$ (move of the controller);
2. $mc = \bot$ and $s^c_1 = s^c_2$ and $s \in Out$ and $a \in \mathcal{T}$ and $inside(\xi_1) \models a$ and $Out_2 = Out_1 \cup \{s'\}$ and $In_2 = In_1$ (move of a passenger outside the train);
3. $mc = \bot$ and $s^c_1 = s^c_2$ and $s \in S_{in}$ and $In_1(s) > 0$ and $a \in \mathcal{T}$ and $inside(\xi_1) \models a$ and $Out_2 = Out_1$ and
   - $In_2(s) = In_1(s) - 1$ and $In_2(s') = In_1(s') + 1$ if $s \neq s'$,
   - $In_2(s) = In_1(s)$ if $s = s'$

   and $In_2(s'') = In_1(s'')$ for all $s'' \in S_{in} \setminus \{s, s'\}$ (move of a passenger in the train);
4. $mc = \bot$ and $s^c_1 = s^c_2$ and $s \in Out$ and $a = \mathbf{E}$ and $inside(\xi_1) < cap + 2$ and $Out_2 = Out_1$ and $In_2(s') = In_1(s') + 1$ and $In_2(s'') = In_1(s'')$ for all $s'' \in S_{in} \setminus \{s'\}$ (a passenger enters the train);
5. $mc = \bot$ and $s^c_1 = s^c_2$ and $s \in S_{in}$ and $In_1(s) > 0$ and $a = \mathbf{Q}$ and $Out_2 = Out_1 \cup \{s_f\}$ and $In_2(s) = In_1(s) - 1$ and $In_2(s'') = In_1(s'')$ for all $s'' \in S_{in} \setminus \{s\}$ (a passenger leaves the train).

We write $\xi_1 \rightsquigarrow \xi_2$ if there exist $\delta \in \Delta$ and $mc = \{\top, \bot\}$ such that $\xi_1 \overset{\delta,mc}{\rightsquigarrow} \xi_2$, and we denote by $\rightsquigarrow^*$ the reflexive and transitive closure of $\rightsquigarrow$.

We shall now see how we can reduce Train-Reachability to a reachability query in the transition system $(\Xi, \rightsquigarrow)$. In other words, we shall prove in which matters our abstraction is sound and complete for Train-Reachability. The results of the two next lemmas need to be combined with the result of Proposition 9 which states that we can restrict our attention to $(cap + 2)$-bounded executions to solve Train-Reachability. First we give the lemma needed to ensure completeness of our abstraction. For this, given an abstract train configuration $\xi = (s^c, Out, In)$, we define $[\![\xi]\!]$, a set of configurations described by ξ. For a train configuration $\theta = (\mathbf{s}, c)$, we let $\theta \in [\![\xi]\!]$ if the following conditions hold:

- $c = inside(\xi)$,
- $\mathbf{s}[1] = s^c$,
- for all $p \in [2, |\theta|]$, if $\mathbf{s}[p] \in S_{out} \cup \{s_f\}$ then $\mathbf{s}[p] \in Out$, and
- $In(s) = |\{p \in [2, |\theta|] \mid \mathbf{s}[p] = s\}|$ for all $s \in S_{in}$.

In other words, the control state of the controller is the same in θ and ξ, the states of the passengers in the train are the same in ξ and θ, and all the states present in θ from passengers outside the train are present in Out. This interpretation of abstract configurations allows us to state our first property.

▶ **Lemma 11.** *Let θ and θ' be two configurations such that θ is initial. If there is a $(cap+2)$-bounded execution from θ to θ' then there exists an abstract train configuration ξ' such that $\theta' \in [\![\xi']\!]$ and $\xi_\iota \rightsquigarrow^* \xi'$.*

To ensure the soundness of our method, for an abstract train configuration $\xi = (s^c, Out, In)$, we need to identify in $[\![\xi]\!]$ the configurations for which all the states in Out are present. We say that a configuration $\theta = (\mathbf{s}, c)$ is a witness for ξ if $\theta \in [\![\xi]\!]$ and, for all $s \in Out$, there exists $p \in [2, |\theta|]$ such that $\mathbf{s}[p] = s$. This new notion combined with the result of the Copycat Lemma 10 allows us to state the following property of our abstraction.

▶ **Lemma 12.** *Let $\xi' \in \Xi$. If $\xi_\iota \rightsquigarrow^* \xi'$ then there exist an initial configuration θ and $\theta' \in [\![\xi']\!]$ such that there is a $(cap + 2)$-bounded execution from θ to θ' and θ' is a witness for ξ'.*

Now to solve Train-Reachability for the train automaton TA and a state $s \in S$, thanks to Proposition 9, we know it is enough to consider only $(cap + 2)$-bounded executions. Lemmas 11 and 12 tell us that we have to seek in the graph $(\Xi, \rightsquigarrow)$ a path between ξ_ι and an abstract train configuration $\xi = (s^c, Out, In)$ such that $s = s^c$ or $s \in Out$ or $In(s) > 0$. Note that by definition $|\Xi| \leq |S^c| \cdot 2^{|S_{out}|+1} \cdot |S_{in}|^{cap+2}$ hence the size of $(\Xi, \rightsquigarrow)$ is exponential in the size of TA and the transition relation $\rightsquigarrow$ can be built on-the-fly (as it is done in its definition). Using that the reachability problem in a graph can be solved in NLogspace, we deduce that we can solve Train-Reachability in NPspace (by solving a reachability query in $(\Xi, \rightsquigarrow)$). Thanks to Savitch's theorem we deduce our Pspace upper bound.

▶ **Theorem 13.** *Train-Reachability is in Pspace.*

5 An algorithm for reachability

In this section, we provide an algorithm to solve Reachability using, as an internal procedure, the algorithm proposed in the previous section for Train-Reachability.

We consider a DMA $\mathcal{A} = (S, \iota, \Delta, F)$. Without loss of generality, we assume that in $\mathcal{A}$ when a process p performs a write action, then it will not do so again until it performs a new action. This restriction makes sense, since when it has written its local value once, it

does not change anything to the behavior of the global system to rewrite it. One can easily modify $\mathcal{A}$ to respect this property by adding a boolean flag to the states which is set to true after a write and set back to false after a new. Moreover, when an edge labelled with write leaves a state while the newly introduced boolean is true, then write is replaced by the test $>_0$ (which will be necessarily evaluated to true since the global memory contains at least the local value of the process). Before presenting our method to solve Reachability, we state a technical lemma similar to the Copycat Lemma 10, but this time for DMA instead of train automata. The idea here is that we can join two distinct executions of the DMA using the fact that in DMA, the precise values of the data written in the global or local memory do not really matter but only the occurrences of the same values are important.

▶ **Lemma 14** (Copycat Lemma II). *If there exists an execution $\gamma_0 \Longrightarrow_{\mathcal{A}}^* \gamma_1$ with γ_0 initial and $\gamma_1 = (\mathbf{s}_1, \boldsymbol{\ell}_1, \mathbf{M}_1)$ and an execution $\gamma_0' \Longrightarrow_{\mathcal{A}}^* \gamma_1'$ with γ_0' initial and $\gamma_1' = (\mathbf{s}_1', \boldsymbol{\ell}_1', \mathbf{M}_1')$, then there exists an execution $\gamma_0'' \Longrightarrow_{\mathcal{A}}^* \gamma_1''$ with γ_0'' initial and such that $|\gamma_1''| = |\gamma_1| + |\gamma_1'|$ and $\gamma_1'' = (\mathbf{s}_1'', \boldsymbol{\ell}_1'', \mathbf{M}_1'')$ with $\mathbf{s}_1''[p] = \mathbf{s}_1[p]$ for all $p \in [1, |\gamma_1|]$ and $\mathbf{s}_1''[|\gamma_1| + p] = \mathbf{s}_1'[p]$ for all $p \in [1, |\gamma_1'|]$.*

As a consequence of this lemma, if at some point we reach a configuration γ_1 in a DMA, we know that any configuration with as many copies as one may desire of the states of γ_1 is reachable. Our algorithm for Reachability then computes, iteratively, the two following subsets of the set of states S:

- $\mathcal{N}\mathsf{ew}$ is the set of reachable states $s \in S$ from which an action new is feasible. Formally, $s \in \mathcal{N}\mathsf{ew}$ if there exist $\gamma, \gamma' \in \mathbb{C}_{\mathcal{A}}$ such that γ is initial and γ' and $\gamma \Longrightarrow_{\mathcal{A}}^* \gamma'$ and $s \in \mathit{states}(\gamma')$ and $(s, \mathsf{new}, u) \in \Delta$ for some $u \in S$.
- $\mathcal{OW}\mathsf{rite}$ is the set of states $s \in S$ that occur in some execution where the process being in s performs new and eventually write (hence the set of states from which a process can overwrite its value in the global memory). Formally, $s \in \mathcal{OW}\mathsf{rite}$ if there exist a run ρ of $\mathcal{A}$ of the form $\gamma_0 \xrightarrow{(\sigma_0,p_0)}_{\mathcal{A}} \gamma_1 \xrightarrow{(\sigma_1,p_1)}_{\mathcal{A}} \gamma_2 \cdots \xrightarrow{(\sigma_\ell,p_\ell)}_{\mathcal{A}} \gamma_{\ell+1}$ and $p \in [1, |\gamma_0|]$ and $0 \leq j < k \leq \ell$ such that $\gamma_j = (\mathbf{s}_j, \boldsymbol{\ell}_j, \mathbf{M}_j)$ with $\mathbf{s}_j[p] = s$ and $(\sigma_j, p_j) = (\mathsf{new}, p)$ and $(\sigma_k, p_k) = (\mathsf{write}, p)$ and, for all $i \in [j+1, \ell-1]$, if $p_i = p$ then $\sigma_i \notin \{\mathsf{new}, \mathsf{write}\}$.

First, note that $\mathcal{OW}\mathsf{rite} \subseteq \mathcal{N}\mathsf{ew}$. We will see now how to compute these two sets of states and how our method exploits the result of the previous section on the train problem. The intuition to link the reachability in DMA with this latter problem is the following: each process in a DMA is associated to a train whose number is the value stored in its local register. When a process writes its value to the global memory, it enters the corresponding train and it stays in it until it overwrites this value by another one (by entering a new train).

We first explain, given two sets of states $\mathcal{N} \subseteq \mathcal{N}\mathsf{ew}$ and $\mathcal{OW} \subseteq \mathcal{OW}\mathsf{rite}$, how to build a train automaton $TA_{\mathrm{N}}(\mathcal{N}, \mathcal{OW})$ to check whether new states can be added to $\mathcal{N}$. We define $TA_{\mathrm{N}}(\mathcal{N}, \mathcal{OW}) = (S_T, \iota_T^c, \iota_T, S_{out}, S_{in}, \Delta_T, s_f)$ with:

- $S_T = (S \times \{out, in\}) \cup \{\iota_T, s_f\}$,
- $S_{out} = (S \times \{out\}) \cup \{\iota_T\}$,
- $S_{in} = S \times \{in\}$,
- $\iota_T^c = (\iota, out)$,
- Δ_T is the set of transitions verifying:
  - $(\iota_T, =_0, (u, out)) \in \Delta_T$ for all $u \in S$ such that there is $(s, \mathsf{new}, u) \in \Delta$ with $s \in \mathcal{N}$,
  - $((s, out), \mathbf{E}, (s', in)) \in \Delta_T$ for all $(s, \mathsf{write}, s') \in \Delta$,
  - $((s, in), \mathbf{Q}, s_f) \in \Delta_T$ for all $s \in \mathcal{OW}$,
  - $((s, out), a, (s', out)), ((s, in), a, (s', in))$ for all $(s, a, s') \in \Delta$ with $a \in \mathcal{T}$.

In a DMA, when a process performs a new, it is always possible that it chooses the initial value of another process that has not been written yet to the global memory. However, for a given value, there is at most one such process since, initially, all the processes have pairwise different values in their local memory. Such a process is represented in $TA_{\mathrm{N}}(\mathcal{N}, \mathcal{OW})$ by the distinguished controller. Hence, the initial state of the controller is (ι, out). All the other processes have to perform a new and are represented by the other passengers. To participate in the train automaton, they have to go through the transitions $(\iota_T, =_0, (u, out))$ such that there is $(s, \mathsf{new}, u) \in \Delta$ with $s \in \mathcal{N}$. The train automaton $TA_{\mathrm{N}}(\mathcal{N}, \mathcal{OW})$ then simulates the DMA with the following rules: When a passenger enters the train with **E**, the associated process writes its value to the current memory, and when he leaves the train with **Q**, the associated process has been able to choose a new value and to write it to the global memory, so intuitively it was in a state of $\mathcal{OW}$.

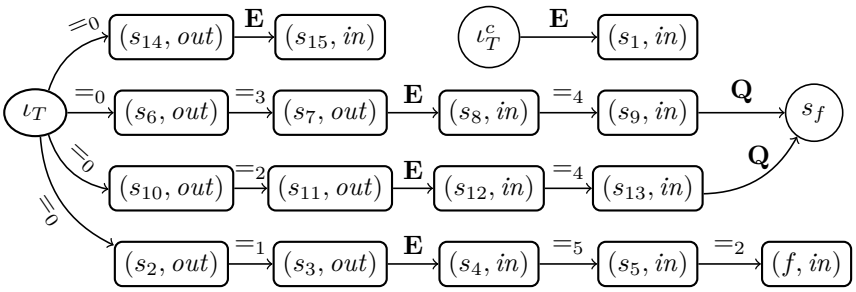


Figure 5 Train Automaton $TA_{\mathrm{N}}(\{\iota, s_9, s_{13}\}, \{\iota, s_9, s_{13}\})$ for the DMA of Figure 1.

▶ **Example 15.** Figure 5 depicts the train automaton $TA_{\mathrm{N}}(\mathcal{N}, \mathcal{OW})$ associated to the DMA of Figure 1 with $\mathcal{N} = \{\iota, s_9, s_{13}\}$ and $\mathcal{OW} = \{\iota, s_9, s_{13}\}$. Thanks to this train automaton, we deduce that f is reachable in the DMA because $(f, in) \in \mathsf{TrainReach}(TA_{\mathrm{N}}(\mathcal{N}, \mathcal{OW}))$. We have indeed the following execution with five passengers (numbered from 1 to 5, where 1 is the controller): First, passengers 2 to 4 move to (s_{10}, out), and passenger 5 moves to (s_2, out). Then, the controller enters the train and arrives in (s_1, in). After that, passenger 5 can go to (s_4, in) entering the train. There are now two passengers in the train, so passengers 2 to 4 can go to (s_{11}, out) and passengers 2 to 3 can enter the train and move to (s_{13}, in) since there will be four passengers in the train. Finally, passenger 4 enters the train. There are now five passengers in the train allowing passenger 5 to move to (s_5, in). After that, passenger 2 in (s_{13}, in) can leave the train, and passenger 4 can move to (s_{13}, in). Now, passengers 3 and 4 from (s_{13}, in)) can leave the train bringing the number of passengers to two which allows passenger 5 to reach (f, in).

Thanks to Lemma 14 (Copycat Lemma) and to the semantics of train automata, we deduce the following:

▶ **Lemma 16.** *Let $\mathcal{N} \subseteq \mathcal{N}\mathsf{ew}$, $\mathcal{OW} \subseteq \mathcal{O}\mathcal{W}\mathsf{rite}$, and $s \in S$. If we have $\{(s, in), (s, out)\} \cap \mathsf{TrainReach}(TA_{\mathrm{N}}(\mathcal{N}, \mathcal{OW})) \neq \emptyset$ and $(s, \mathsf{new}, u) \in \Delta$ for some $u \in S$, then $s \in \mathcal{N}\mathsf{ew}$.*

Hence this last lemma allows us to add new states from $\mathcal{N}\mathsf{ew}$ to $\mathcal{N}$. We will now see how to increase the set of states $\mathcal{OW}$. The idea is similar but we give as input a state sn in $\mathcal{N}$ from which we want to check whether an action write can be reached. In the train automaton, we hence have to check which states are reachable from this state sn. For this matter, we use an extra symbol, $\top$ or $\bot$, to track the path coming from sn (this symbol equals $\top$ when the state is reachable from sn). Given two sets of states $\mathcal{N} \subseteq \mathcal{N}\mathsf{ew}$ and $\mathcal{OW} \subseteq \mathcal{O}\mathcal{W}\mathsf{rite}$ and

$sn \in \mathcal{N}$, we build a train automaton $TA_{\mathrm{OW}}(\mathcal{N}, \mathcal{OW}, sn)$ to check whether sn can be added to $\mathcal{OW}$. We let $TA_{\mathrm{OW}}(\mathcal{N}, \mathcal{OW}, sn) = (S_T, \iota_T^c, \iota_T, S_{out}, S_{in}, \Delta_T, s_f)$ with:

- $S_T = (S \times \{out, in\} \times \{\top, \bot\}) \cup \{\iota_T, s_f\}$
- $S_{out} = (S \times \{out\} \times \{\top, \bot\}) \cup \{\iota_T\}$,
- $S_{in} = S \times \{in\} \times \{\top, \bot\}$,
- $\iota_T^c = (\iota, out, \bot)$,
- Δ_T is the set of transitions verifying:
  - $(\iota_T, =_0, (u, out, \bot)) \in \Delta_T$ for all $u \in S$ such that there is $(s, \mathsf{new}, u) \in \Delta$ with $s \in \mathcal{N}$,
  - $(\iota_T, =_0, (u, out, \top)) \in \Delta_T$ for all $u \in S$ such that $(sn, \mathsf{new}, u) \in \Delta$,
  - $((s, out, v), \mathbf{E}, (s', in, v)) \in \Delta_T$ for all $(s, \mathsf{write}, s') \in \Delta$ and $v \in \{\top, \bot\}$,
  - $((s, in, v), \mathbf{Q}, s_f) \in \Delta_T$ for all $s \in \mathcal{OW}$ and $v \in \{\top, \bot\}$,
  - $((s, out, v), a, (s', out, v)), ((s, in, v), a, (s', in, v))$ for all $(s, a, s') \in \Delta$ with $a \in \mathcal{T}$ and all $v \in \{\top, \bot\}$.

Hence in this train automaton, if a state $(s, in, \top)$ or $(s, out, \top)$ is reached, the passenger reaching this state necessarily went through the state $(sn, out, \top)$. We have the following result whose correcteness can be proved the same way as for Lemma 16.

▶ **Lemma 17.** *Let $\mathcal{N} \subseteq \mathcal{N}\mathsf{ew}$, $\mathcal{OW} \subseteq \mathcal{OW}\mathsf{rite}$, and $sn \in \mathcal{N}$. If there exists $s \in S$ such that $(s, out, \top) \in \mathsf{TrainReach}(TA_{\mathrm{OW}}(\mathcal{N}, \mathcal{OW}, sn))$ and such that $(s, \mathsf{write}, u) \in \Delta$ for some $u \in S$, then $sn \in \mathcal{OW}\mathsf{rite}$.*

These two last lemmas give us a technique to compute the sets $\mathcal{N}\mathsf{ew}$ and $\mathcal{OW}\mathsf{rite}$. We present a procedure that computes iteratively two families of sets of states $(\mathcal{N}_i)_{i\in\mathbb{N}}$ and $(\mathcal{OW}_i)_{i\in\mathbb{N}}$ such that $\mathcal{N}_i \subseteq \mathcal{N}_{i+1} \subseteq \mathcal{N}\mathsf{ew}$ and $\mathcal{OW}_i \subseteq \mathcal{OW}_{i+1} \subseteq \mathcal{OW}\mathsf{rite}$ for all $i \in \mathbb{N}$. We set $\mathcal{N}_0 = \mathcal{OW}_0 = \emptyset$ and, for all $i \in \mathbb{N}$:

- $\mathcal{N}_{i+1} = \mathcal{N}_i \cup \left\{ s \in S \;\middle|\; \begin{array}{l} \{(s, in), (s, out)\} \cap \mathsf{TrainReach}(TA_{\mathrm{N}}(\mathcal{N}_i, \mathcal{OW}_i)) \neq \emptyset \text{ and} \\ (s, \mathsf{new}, u) \in \Delta \text{ for some } u \in S \end{array} \right\}$
- $\mathcal{OW}_{i+1} = \mathcal{OW}_i \cup \left\{ sn \in \mathcal{N}_{i+1} \;\middle|\; \begin{array}{l} \exists s \in S. \\ (s, out, \top) \in \mathsf{TrainReach}(TA_{\mathrm{OW}}(\mathcal{N}_{i+1}, \mathcal{OW}_i, sn)) \text{ and} \\ \quad (s, \mathsf{write}, u) \in \Delta \text{ for some } u \in S \end{array} \right\}$

Note that, since the set of states S is finite, these computations terminate and, thanks to Theorem 13, we know they are in PSPACE. We define $\mathcal{N} = \bigcup_{i\in\mathbb{N}} \mathcal{N}_i$ and $\mathcal{OW} = \bigcup_{i\in\mathbb{N}} \mathcal{OW}_i$. Due to Lemmas 16 and 17, we have $\mathcal{N} \subseteq \mathcal{N}\mathsf{ew}$ and $\mathcal{OW} \subseteq \mathcal{OW}\mathsf{rite}$. We can also obtain the inclusion in the other directions by reasoning by induction on the length of the executions of the DMA and looking at the processes that can create a new value or can overwrite their value in the global memory in such executions.

▶ **Lemma 18.** *We have $\mathcal{N} = \mathcal{N}\mathsf{ew}$ and $\mathcal{OW} = \mathcal{OW}\mathsf{rite}$.*

Now, to conclude, we can assume w.l.o.g. that, from each of the final states s in F, there is a transition (s, new, s') in Δ (if not we can add one) and hence solving REACHABILITY amounts at verifying whether $F \cap \mathcal{N} \neq \emptyset$. Since, as said earlier, $\mathcal{N}$ and $\mathcal{OW}$ can be computed in PSPACE, this allows us to deduce the following theorem:

▶ **Theorem 19.** *REACHABILITY is in PSPACE.*

6 Conclusion

We have shown that the control-state reachability problem for DMAs is in PSPACE when the number of processes is a parameter and is PSPACE-complete when this number is fixed. The upper-bound for the parameterized case is obtained thanks to an algorithm which uses as a sub-routine a polynomial-space solution for the control-state reachability in train automata. If we could find a better complexity bound, such as P or NP, for TRAIN-REACHABILITY, this bound will also apply to REACHABILITY in DMAs. Similarly, if we find another algorithm to solve REACHABILITY in DMAs with a better upper bound, this would lead to a better solution for TRAIN-REACHABILITY (which can easily be encoded into REACHABILITY for DMAs). In fact, we currently do not have any lower bound for these two problems and the proof to obtain the lower bound for FIXED-REACHABILITY crucially depends on the fact that we know the number of involved processes. In the future, we plan to further study the TRAIN-REACHABILITY problem and some of its extensions to see how the reasoning presented here can be applied to verify concrete distributed algorithms.

References

1. C. Aiswarya, Benedikt Bollig, and Paul Gastin. An automata-theoretic approach to the verification of distributed algorithms. *Inf. Comput.*, 259(Part 3):305–327, 2018.
2. Henrik Björklund and Thomas Schwentick. On notions of regularity for data languages. In Erzsébet Csuhaj-Varjú and Zoltán Ésik, editors, *Fundamentals of Computation Theory, 16th International Symposium, FCT 2007*, volume 4639 of *Lecture Notes in Computer Science*, pages 88–99. Springer, 2007.
3. Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. *Decidability of Parameterized Verification*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.
4. Mikolaj Bojanczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 12(4):27:1–27:26, 2011.
5. Benedikt Bollig, Patricia Bouyer, and Fabian Reiter. Identifiers in registers - describing network algorithms with logic. In Mikolaj Bojanczyk and Alex Simpson, editors, *Foundations of Software Science and Computation Structures - 22nd International Conference, FOSSACS 2019*, volume 11425 of *Lecture Notes in Computer Science*, pages 115–132. Springer, 2019.
6. Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. The renaming problem in shared memory systems: An introduction. *Comput. Sci. Rev.*, 5(3):229–251, 2011.
7. Stéphane Demri and Ranko Lazic. LTL with the freeze quantifier and register automata. *ACM Trans. Comput. Log.*, 10(3):16:1–16:30, 2009.
8. Javier Esparza. Keeping a crowd safe: On the complexity of parameterized verification (invited talk). In Ernst W. Mayr and Natacha Portier, editors, *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*, volume 25 of *LIPIcs*, pages 1–10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014.
9. Wan Fokkink. *Distributed Algorithms: An Intuitive Approach*. MIT Press, 2013.
10. Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.
11. Dexter Kozen. Lower bounds for natural proof systems. In *FOCS'77*, pages 254–266. IEEE Computer Society, 1977.
12. Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.
13. Nikos Tzevelekos. Fresh-register automata. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011*, pages 295–306. ACM, 2011.